Dynamic Set/Vector Implementation Analysis

Author: syscl/Yating Zhou

November 22, 2018

This paper discussed some potential performance improvement invokes with *realloc* for dynamic growth set. Suppose the data structure D have a single *append* (i.e. *emplace_back*) as the data structure's operations. What we want is to reduce the overall runtime of these operations. *append*(x) operation will attach an element x at the end of a given dynamic set D, and resize will happen if the space of D is not enough.

If there's still empty space of D, *append*(x) will take constant runtime (i.e. $O(1)$). Otherwise, *realloc* is invoked to allocate enough space for our data structure then append the element to the new construct D. We will show the strategy of how *realloc* new space will greatly affect the overall runtime appending an element. Let the size of D to be m, the usual way to resize the D with a growth factor of $\lambda > 1$; i.e., whenever you'd append an element into D without there being room, it would expand the current capacity to λ times of its original size. For example the *gcc* compiler use a fixed $\lambda = 2$ for the D to resize. If the size of an element x is $\alpha > 0$, D will be resized when there's λ^k $(k \geq 0)$ elements, suppose we have n resize on D, then this will require $\alpha\lambda^n$ space, to enable cache-friendly properties (i.e. reuse old elements without involving deep copy), it is neccessary for the following inequality hold

$$\sum_{i=0}^{n-1} \alpha\lambda^n = \alpha \sum_{i=0}^{n-1} \lambda \geq \alpha\lambda^n \tag{1}$$

(1) says precisely that if **all** previous allocated space will result in D that is greater or equal to current new resize space, we can then make the memory managemenet or cache hit more friendly (i.e. less page fault). The left hand side of (1) is the sum of geometric progression, so that (1) turns out to be

$$\lambda^{n+1} - 2\lambda^n + 1 \leq 0 \tag{2}$$

or

$$\lambda^n(\lambda - 2) + 1 \leq 0 \tag{3}$$

By (2) or (3), if $\lambda \geq 2$, the left hand size of (3) will become $\lambda^n(\lambda - 2) + 1 \geq 1$, which indicates we can not enable cache-friendly memory management under this case. So now we turns to focus on $1 < \lambda < 2$ of (2). What we care is (2) and (3) holds after several calls of *realloc*, in other words, if there exists an integer N such that for $n \geq N$, (2) holds, then we can enable cache-friendly memory management, so now the problem turns out to be choosing the proper λ. By (2) and $1 < \lambda < 2$, we have

$$n \geq -\frac{ln(2-\lambda)}{ln(\lambda)} \tag{4}$$

By (4) let

$$f(x) = -\frac{ln(2-x)}{ln(x)} \tag{5}$$

(4) and (5) give us

$$n \geq f(\lambda) \tag{6}$$

It's clear that if after several N, if $n \geq N = f(\lambda)$, then we can easily improve the overall runtime of our data structure, and we don't want the N turns out to be too large obviously. So we have to explore the properties of the function $f(x)$.

$$f^{'}(x) = \frac{xln(x) + (2-x)ln(2-x)}{x(2-x)ln^2(x)} \tag{7}$$

Let

$$g(x) = xln(x) + (2-x)ln(2-x) \tag{8}$$

By (8)

$$g^{'}(x) = ln\frac{1}{\frac{2}{x} - 1} + \frac{2}{2-x} \tag{9}$$

Since $1 < x < 2$, then $0 < \frac{2}{x} - 1 < 1$, along with (9) gives us

$$g^{'}(x) > 0 \tag{10}$$

(10) indicates that $g(x)$ is increasing in $(1, 2)$, so

$$g(x) > g(1) = 0 \tag{11}$$

(7), (8) and (11) give us

$$f^{'}(x) > 0 \tag{12}$$

Again, (12) implies that $f(x)$ is an increasing function in interval $(1, 2)$, so that we can determine how to choose proper x (or λ) for our data set. If $x \to 2^-$, by (8)

$$\lim_{x \to 2^-} f(x) = \lim_{x \to 2^-} -\frac{ln(2-x)}{ln(x)} = +\infty \tag{13}$$

(14) tells us that if we choose x close to 2, it will result in our data structure hard to enable the cache-friendly memory management, which is not a good choice. If $x \to 1^+$, (8) with L'Hospital's rule give us

$$\lim_{x \to 1^+} f(x) = \lim_{x \to 1^+} -\frac{ln(2-x)}{ln(x)} \tag{14}$$

$$= \lim_{x \to 1^+} \frac{x}{2-x} = 1 \tag{15}$$

(15) gives us the fact if x is small enough (close to 1), it is really easy for us to enable the memory management, however this will invokes in too many resize (1 per call of *append*), in other words, this yiels in $O(n)$ runtime of both *append* and *realloc*. If we choose any λ between 1 and 2, we can then improve the overall runtime complexity. Before we finish the assertion, we have to prove the following lemma.

Lemma1: The runtime complexity of resize of a fixed factor $1 < \lambda < 2$ is proportion to the runtime complexity of double resize strategy with λ.

Proof. If we have $n = 2^k$ elements to insert, suppose we have φ calls for a given λ resize strategy. Then it is obvious we have $k = log(n)$ calls if we choose double size strategy, we will have $2^k = \lambda^\varphi$ calls for realloc if we choose λ size strategy. This gives us

$$\varphi = -k\ log(\lambda) \tag{16}$$

(16) is what we want because $O(\varphi) = O(k)$. □

By (12), (13), (15) and lemma1, we have the following fact:

- Choose λ in interval $(1, 2)$ enables the cache-friendly memory management
- $\varphi = -k\ log(\lambda)$ implies that the realloc number will be portional to $-log(\lambda)$, so we have to "balance out" the λ in case the $-log(\lambda)$ not become too large.

For example, choosing the middle point of 1 and 2 gives us the overall calls for n insertion to be $O(\varphi) = O(1.71k) = O(1.71\ log(n)) = O(log(n))$, a trivial $\lambda = 1$ gives us the overall calls to be $O(n)$ and if $\lambda = 2$, the overall runtime will be $O(log(n))$ but don't forget there will be $\sum_{i=0}^{k} i = \frac{k(k-1)}{2} = O(log^2(n))$ involves for deep copy every time.

References

[1] https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md